**Analysis and Improvement of Current MASS-based GIS**

**Shahruz Mannan**

Capstone Term Report Au23 submitted   
in partial fulfilment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

## University of Washington 12/13/2023

**Project Committee:**

Dr. Munehiro Fukuda, Committee Chair

Dr. Michael Stiber, Committee Member

Dr. Dong Si, Committee Member

1. **Project Overview**

As geospatial data or geodata becomes more complex, the need for performance improvements and data handling enhancements in the geospatial information systems (GIS) becomes increasingly important. This could be achieved for example with the use of parallel computing techniques. The distribution of computations across multiple processors or computers can significantly enhance the performance and increase the speed of data processing of geodata. Multi-Agent Spatial Simulation (MASS) [1] is a parallel-computing library which is designed to support large-scale simulation models, such as agent-based models and cellular automata. MASS provides a scalable platform for developing and running models that involve large numbers of autonomous agents acting in a spatial context. This library allows these models to be parallelized across multiple processors, which significantly speeds up their execution. Figure 1 illustrates a typical parallel execution process using the MASS library. In this GIS context, performance is usually gauged by the effectiveness of the system's CPU scalability and the amount of geodata it can support. Thus, MASS can be a viable option for these GIS improvements.

This project will delve into two main areas. The first focus area is the improvement of the performance of the MASS-based GIS by exploring its CPU scalability. Current Geospace + MASS-Java implementations which previous students have implemented will be compared with Message Passing Interface (MPI) implementations. Depending on the outcomes of these comparisons, the project will either target the enhancement of CPU scalability for GIS parallelization or pivot towards spatial scalability. This shift in focus would assess the system's ability to handle larger GIS space with increased CPUs, as opposed to purely looking at speed improvements.

Given that the current implementation of MASS-based GIS predominantly focuses on vector data, this project additionally proposes to incorporate and analyze raster data to MASS Places. The objective is to parallelize GIS attribute/spatial queries using MASS Places and Agents, thereby broadening the system's data handling capabilities. The selection of these queries will be guided by two main criteria: computational intensity and relevance to real-world applications. Operations that are computationally intensive and could significantly benefit from parallelization will be prioritized. In addition, operations commonly used in real-world GIS applications will be considered to ensure the practical relevance of the project. For selecting GIS attribute/spatial queries, this project will reference Raghavendra's (2023) white paper on Agent-based GIS Queries."

By integrating these improvements, the project aims to enhance the computational performance of MASS-based GIS, while widening its scope to handle raster data as well. Through these efforts, this project will provide a significant contribution to the advancement of geospatial data handling and processing capabilities.

A diagram of a system

Description automatically generated  
*Figure 1. Parallel execution using MASS [2]*

1. **Goals & Criteria**
   1. **Goals**

The primary goal of this project is to enhance the computational performance and data handling capabilities of the MASS-based GIS. The main objectives are shown below:

* Goal 1: Develop a foundational understanding of the current MASS-based GIS performance by studying the existing Geospace + MASS-Java implementation. This analysis will serve as a benchmark for subsequent performance improvements. It will include an evaluation of the current CPU scalability and vector data handling capabilities, thereby establishing a starting point for performance and data handling enhancements.
* Goal 2: Compare the current MASS-based GIS implementation with MPI implementations to understand their differences in CPU scalability. This comparison will identify potential paths for performance improvement in MASS-based GIS.
* Goal 3: Depending on the analysis from Goal 2, if the MPI implementation shows CPU scalability, I will work on improving the CPU scalability of MASS-based GIS. If the implementation does not show CPU scalability, I will either try to continue to work on improving the CPU scalability by tackling the obstacles or shift the focus to spatial scalability. If spatial scalability becomes the focus, I will assess the system's capacity to support larger GIS spaces with more CPUs, aiming to enhance the efficiency of processing larger volumes of geospatial data.
* Goal 4: Integrate raster data handling into MASS-based GIS, currently focusing on vector data. This will involve mapping raster data to MASS Places and parallelizing some GIS attribute/spatial queries with MASS Places and Agents. This goal will extend the capabilities of the system to process and analyze a broader range of geospatial data.
* Goal 5: Conduct a comprehensive evaluation of MASS-based GIS after implementing the above improvements. This evaluation will examine the execution performance and programmability analysis of the improved system.

1. **This quarter’s achievements**
   1. **Review Literature and Understand the System**

A proportion of the efforts in this quarter were dedicated to reviewing literature and understanding the current MASS-GIS system. This involved understanding how MASS works, how the MASS library is used with a GIS system and exploring the existing implementations and benchmarks in the MASS-GIS system. This led to the discovery of two aspects of the current implementation: The integration of Contextual Query Language (CQL) [3] with MASS and Computational Geometry [15] with MASS. CQL is a query language used for retrieving information from information systems. Particularly, in this scenario, it is used to query information that involves spatial or attribute data. Computational Geometry is a field dedicated to solving problems related to geometric properties such as points, polygons, and lines. Figure 2. shows the two categories of the parallelization of GIS queries.

**A diagram of a computer program

Description automatically generated with medium confidence**

*Figure 2. Parallel GIS queries in current MASS-GIS system [4]*

The integration of CQL with MASS enables executing GIS queries in parallel which guarantees faster query executions and more effective data handling. The CQL queries are executed data partitions maintained on individual MASS Places. The query results are then sent to the master node. With the current implementation, a few examples of spatial and attribute queries were executed for benchmarking. One of the spatial queries was to find all cities located within a 100 km radius of Seattle. An example of the attribute query was to find all cities with a population greater than a specific threshold. These benchmarks showed great CPU scalability and a noticeable reduction in execution time as the number of computing nodes increased [4].

The objective for combining Computational Geometry with MASS was to be able to execute specific types of GIS queries in the MASS-GIS system. These Computational Geometry problems implemented are Closest Pair of Points (CPP) [16], Range Search (RS) [17], and Minimum Spanning Tree (MST) [18]. The first Computational Geometry problem, Closest Pair of Points finds the closest pair of points from a given set. Using CPP with MASS allows the processing of large amounts of data efficiently to find the closest pairs. One of the GIS queries executed for benchmarking was “Find the closest places of fire occurrences given a coordinate position.” The benchmarks resulted in a significant decrease in query execution time as more computing nodes were used [4].

The final Computational Geometry problem, the Minimum Spanning Tree in the GIS concept is used for connecting all data points together with the shortest distances without any cycles. This connected graph is then used for finding the shortest path between two geographical coordinates. For benchmarking, a spatial query to find the shortest path from a given source to a destination on a railroad dataset [5] was executed. The query was executed the fastest when 3 agents were used. Using more agents resulted in more time being spent on exchanging information between the agents and killing the agents. In addition, the more computing nodes were used the more time it took for the agents to migrate. Cui (2022) additionally implemented a Minimum Spanning Tree with MASS. These benchmark results showed that the execution speed increases with the number of computing nodes because of the memory overhead and the communication time between the computing nodes. This most likely means that MST is not parallelizable with MASS. From these observations, I was able to conclude on what kinds of improvements I could potentially work on.

**3.2. Selection of GIS Attribute/Spatial Queries**

In this phase of the project, the focus was on determining what kinds of improvements could be implemented for the MASS-GIS system. Based on the current MASS-GIS implementation, I decided to focus on the Computational Geometry problems with MASS part. I wanted to improve especially the Range Search problem and the Minimum Spanning Tree problem because both have great relevance to the real-world GIS applications. Additionally, I wanted to implement more different types of Computational Geometry problems with MASS. These implementations would add more GIS queries which could be used with the MASS-GIS system and solve more relevant real-world problems. Below are the computational geometry problems I decided to work on in this quarter.

* Range Search (RS)
* Convex Hull (CH)
* Largest Empty Circle (LEC)
* Euclidean Shortest Path (ESP)

I chose to improve the Range Search implementation because its ability to find features within a specific range is a common requirement in geospatial analysis. In the ”Review Literature and Understand the System” subsection, I mentioned a GIS query for this problem. Other potential GIS queries with the Range Search problem are: "List all schools within a 100-mile radius of Location Y." for educational resource allocation and "Find all gas stations located within a 10-mile radius of a highway exit." for route planning.

Convex Hull problem was chosen because it gives the ability to execute different types of GIS queries. This problem can determine the outer boundary of a set of data points which is useful for GIS queries such as “Find the boundary of the forest region considering all the tree locations” and “Calculate the boundary of the area affected by the oil spill in region A."

The next computational geometry problem I wanted to focus on was the Largest Empty Circle problem. With LEC implementation, we can identify the largest empty space within a set of points that do not have any points inside. LEC is especially good for GIS queries such as “Identify the largest area not covered by any emergency service in a city”. This GIS query would be very valuable for building new emergency facilities.

Lastly, I chose to work on the Euclidean Shortest Path Problem. This problem can help find the shortest path between two points, considering obstacles. This problem handles similar GIS queries as the Minimum Spanning Tree. Implementing the shortest path functionality is highly beneficial in the GIS context because it is crucial in many applications such as urban planning and emergency response. In addition, since MST was not parallelizable with MASS, I wanted to research whether ESP is parallelizable with MASS.

The selection of these Computational Geometry problems gives us an opportunity to enhance the functionality and applicability of MASS-based GIS. This would show the robustness and the diverse geospatial analyses MASS-GIS could do and show applicability in more real-world problems.

* 1. **MPI Comparison**

Lastly, in this quarter I worked on implementing the selected Computational Geometry problems with MPI Java to assess their CPU scalability and how well these problems are parallelizable. This portion was crucial to understand how these problems could be tackled in a distributed computing environment. Each subsection briefly describes how the problems were implemented and how these problems were parallelized with MPI.

* + 1. ***Range Search MPI***

As mentioned in the “Review Literature and Understand the System” section, the Range Search problem is used to identify all points within a specified range from a set of points. The previous MASS implementation of Range Search used a K-d Tree data structure to organize the points and search the Tree for the points that are in the range. Thus, I wanted to use the K-d Tree data structure to check its construction time because based on the previous benchmark results, the tree construction took mostly all the time.

This implementation starts by reading the data points from a CSV file & and partitioning the data points equally to all the computing nodes. Each computing node then constructs its own K-d Trees and does a search on the sub-K-d Trees to find the points that are in the specified range. K-d Trees are binary trees where every node is a k-dimensional point (in our case, a 2-dimensional point). The tree is built recursively by selecting a dimension, sorting the points based on that dimension, and finding the middle point. This middle point gets added to the node and the rest of the points are split into left subtree and right subtree. In this scenario, in every even layer (root being layer 0) the x coordinate was the dominating dimension and in every odd layer, the y coordinate was the dominating dimension. I used the Quickselect algorithm [14] to find the middle point in an average case of O(N) time. With the Quickselect algorithm, the K-d Tree construction takes O(NlogN) time where N is the size of the data partition. Querying the K-d Tree takes O(N^(1-1/k) + m) time, where m is the number of reported points [17]. Figure 8. shows an example of a K-d Tree. After all the computing nodes have found their points in range, I use the MPI\_Gather MPI function to gather all these points in the master rank 0. Finally, the points are written to an output text file.

A diagram of a tree

Description automatically generated with medium confidence

*Figure 8. Example of K-d Tree [6]*

* + 1. ***Convex Hull MPI***

The Convex Hull problem determines the smallest outer boundary of a set of points. I started this problem similarly by reading the data points and partitioning the data evenly into slices for all the computing nodes. In this case, I sorted the points before partitioning the data, so all the partitioned data points would be near each other. To compute the convex hull points. I used Andrew's monotone chain algorithm [7]. How this algorithm works is it constructs the upper hull and lower hull separately. The upper hull computation starts from the most left point and then checks for each point if a left turn is made with the last two upper hull points and the current point. If this happens the second last point is not part of the convex hull and needs to be removed. This is done until no left turns are found and then this current point is added to the hull. The lower hull is computed in a similar manner but starting from the most right point and a check for a right turn happens instead for a left turn. Lastly, the upper and the lower hulls are combined. The right and left turns are computed by calculating the determinant of the 3x3 matrix shown in Equation 1. The time complexity of Andrew’s monotone chain algorithm is O(NlogN) where N is the size of the data partition.

*Equation 1. Determinant formula [8]*

After creating the partial hulls in every computing node, MPI\_Send and MPI\_Recv functions are used to send and receive hull points from other ranks to combine the hulls in a divide-and-conquer-like manner until the final convex hull is in the master rank 0. Two partial convex hulls are merged by finding the upper and lower tangent lines. The tangent lines are calculated in a similar manner by checking for right and left turns until the lines don’t intersect any other points in the hull. After the tangent lines are found, the data points that are between these tangent lines get removed. Figure 9. displays finding the two tangent lines between two different Convex Hulls.

A drawing of a geometric figure with Silverstone Circuit in the background

Description automatically generated

*Figure 9. Finding tangent lines for partial Convex Hulls [7]*

Another algorithm I used to compute the Convex Hull was the Graham scan [9]. How this differs from the other algorithm is we find the lowest data point (smallest y coordinate) and sort the data points on the polar angle based on the reference point’s x-axis. The points are then iterated in a counterclockwise measure checking for a left turn. This algorithm’s time complexity is also O(NlogN). The execution performance between these two algorithms is discussed in the Results section.

* + 1. ***Largest Empty Circle MPI***

This computational geometry problem identifies the largest empty space within a set of points that do not hold any points inside. For this implementation, I created a Voronoi Diagram [10] to help find the largest empty circle. A circle’s center point is either a Voronoi vertex or an intersection point between the data points’ Convex Hull and a Voronoi Edge. I started the implementation in a similar manner by reading the data points. Next, I created the Voronoi Diagram using the Fortune Sweep algorithm [11] sequentially in the master rank 0. In the algorithm, there is a horizontal sweep line that splits the plane into two regions: visited points and not visited points. The visited region contains the partially constructed Voronoi Diagram. As the sweep line moves vertically (top to bottom), it maintains a beach line which evolves as the sweep line moves. When a new point is encountered, the beach line is updated to include a new parabola corresponding to this new point, and a new Voronoi edge is added. This is called a site event. Another event exists called a circle event, which is the bottom-most point of a circle that touches three site points. When a circle event is visited, the corresponding parabola is removed, and a Voronoi vertex is added. This is an O(NlogN) algorithm, where N is the number of points. The efficiency comes from the use of the priority queue to handle which point to process next and the binary search tree to keep track of the parabola on the beach line. Figure 10. displays a circle event happening in a Fortune Sweep execution. After generating the Voronoi diagram, I computed the Convex Hull points from the data points with the Andrew's monotone chain algorithm I implemented. Altogether, the time complexity to get the Voronoi vertices and the convex hull points was O(NlogN + NlogN = NlogN).

A graph of a triangle with lines and circles

Description automatically generated

*Figure 10. Fortune’s Sweep [12]*

After computing the Voronoi vertices and the Convex Hull points, I split the Voronoi vertices, Voronoi edges, and the Convex Hull points and sent these partitions with MPI to the computing nodes. These nodes then compute the intersection points between the partition Voronoi Edges and the Convex Hull edges to get the remaining partition intersection points. Lastly, the computing nodes iterate through these potential center points and calculate the radius to their closest original data point. Finally, the local largest empty circle’s information (radius, center point) is sent back to the master rank 0. This part is the most computationally intensive part which is the reason I wanted to parallelize this segment of the implementation. Finding the convex hull intersection is O (E\*C) where E is the number of partition edges, and C is the convex hull points. Finding the largest empty circle is bound to O(N\*M) where N is the number of all data points, and M is the number of all potential center points in a computing node. Figure 11. describes a Voronoi Diagram and the largest empty circle from these data points. In this figure the red dots are the original data points, and the blue dots are the largest empty circle’s potential center points.

A network with dots and lines

Description automatically generated

*Figure 11. Voronoi Diagram for 50 points and the Largest Empty Circle*

* + 1. ***Euclidean Shortest Path MPI***

This problem is for finding the shortest path from start to destination through polygon obstacles. This implementation is not yet finished because of a logic-breaking bug that I am currently trying to fix. The algorithm I use for this problem is creating a visibility graph with a rotational sweep with Lee's Visibility Graph Algorithm [13]. This algorithm is O(N^2logN) which is better than the naïve version of O(N^3). After creating the visibility graph, Dijkstra’s path-finding algorithm [19] is used to find the shortest path. A visibility graph is a graph where the vertices are the corners of the obstacles and have an edge to other vertices that do not go through the obstacles. Essentially, how this algorithm works is for each point we have a line rotating around 360 degrees to check the visibility to the other points. As it rotates, the points are sorted by the angle can keeps track of the obstacle edges it intersects. If there is no obstacle edge blocking to a point, an edge is added between these points. Figure 12. shows my current implementation of this algorithm. There are a few edges that go through obstacles. As for parallelizing this problem, I have two potential approaches. One is partitioning the data points to computing nodes and each node would find the visibility graph edges for those data points and send the edges back to the master rank to execute Dijkstra’s algorithm. Another approach is to use Parallel Dijkstra’s algorithm which could get quite complex.

A graph of a geometric figure

Description automatically generated with medium confidence

*Figure 12. Visibility Graph with rotational sweep*

1. **Results**

The Range Search benchmark was conducted with my own created dataset of 1 million randomized (latitude, longitude) coordinates. Figure 13. shows 4 different execution times with different numbers of computing nodes. The times are as follows from left to right: average of all K-d Tree construction time, average query time, total algorithm time (including MPI communications), and total elapsed time (including data read, data partitioning & data write). The table presents that the Range Search problem is CPU scalable and the use of MPI parallelizes the Range Search algorithm. The total time after 7 computing nodes starts stabilizing. The total elapsed time with a higher number of nodes suggests that communication overhead becomes more significant. Compared to the current MASS-GIS system’s benchmarks, the K-d Tree constructions take less time even though using more data points. This suggests that the current Range Search MASS-GIS implementation has the potential to be improved.

*Figure 13. Benchmark results for Range Search MPI*

The Convex Hull problem was benchmarked with 4 million randomized points .txt file. Figure 14. and Figure 15. only have two execution times. The compute time means the whole convex hull algorithm (including sorting the data), and the total elapsed time is the whole program’s execution time. Figure 14. and Figure 15. show that the Convex Hull problem is slightly CPU scalable, but it is not much. Andrew’s monotone chain algorithm’s execution times are better overall but they don’t really change much. With the Graham scan’s execution times, the performance gains can be seen better. Both get worse performance with 3 nodes. This is because the convex hulls are merged in a divide-and-conquer way and the 3rd computing node needs to wait for the 1st and 2nd nodes to merge their hulls which causes communication overhead. As more nodes are used, the execution times become similar because the communication overhead “caps out”, thus taking around the same amount of time to merge these hulls together.

As for the difference between these two algorithms, the first one has fewer comparisons between points because two separate loops are executed to check the orientation of the points. With the Graham scan, more nodes mean less data within the ranks and the one loop that goes through all the partition points and makes fewer comparisons between the points. This is most likely the reason Andrew’s monotone chain algorithm has better execution time in general with a large dataset. Nevertheless, implementing Convex Hull for MASS-GIS enables new GIS queries that can be used, and with the dataset being 4M, the execution times are not terrible. For implementing the Convex Hull problem with MASS in the future, I will use Andrew’s monotone chain algorithm. Although, this algorithm does not show parallelism with 20 nodes, it is comparable with Graham scan algorithm.

*Figure 14. Benchmark results for Convex Hull using Andrew’s monotone chain algorithm.*

*Figure 15. Benchmark results for Convex Hull using Graham scan algorithm.*

The Largest Empty Circle was benchmarked with a 50,000 randomized points dataset. The points are normalized with a specified height and width. This makes it so that all the points are defined inside the specified rectangle and the largest empty circle’s center point is inside the rectangle. In this benchmark, the compute time holds the largest empty circle algorithm time, and the total time additionally includes the data read. Figure 16. displays that the problem is highly scalable as more computing nodes are used. The performance improvement slows down after 14 nodes meaning communication overhead starts being a major factor in the execution. The optimal range seems to be around 12-14 nodes. The compute times and the total times are almost the same because reading the data does not take time at all. In addition, this figure shows how efficient the Fortune Sweep algorithm is (bound to O(NlogN)). Most of the computing time is spent finding the Convex Hull and Voronoi edge intersections and calculating the largest empty circle.

*Figure 16. Benchmark results for Largest Empty Circle MPI*

1. **Next quarters plan**

For the next quarter, the highest priority is to finish the Euclidean Shortest Path MPI as soon as possible and benchmark with 20 nodes. Looking at the project plan, I will start working on the MASS implementations for computational geometry problems. I have already started researching raster data. I will continue researching more on how I could use raster data effectively with my MPI implementations first and then with the MASS implementations. After the initial MASS implementations are completed, I will work on applying these implementations with actual GIS queries to do actual real-world geospatial analyses.

1. **Summary**

This project focuses on the computational performance and data handling capabilities of a MASS-based GIS system. The main objectives are improving CPU scalability, integrating raster data, and parallelizing GIS queries. During this quarter I examined the existing MASS-GIS and focused on its implementations with Contextual Query Language (CQL) and Computational Geometry problems. I developed MPI implementations for several computational geometry problems: Range Search (RS), Convex Hull (CH), Largest Empty Circle (LEC), and Euclidean Shortest Path (ESP). The goal was to examine their CPU scalability and parallelizability.

Range search demonstrated CPU scalability with improvements in execution times. After 7 nodes, due to the communication overhead, the execution times became constant. For Convex Hull, both Graham Scan and Andrew’s Monotone Chain algorithms showed CPU scalability. However, as more nodes were used the execution times became similar because the communication overhead “capped out”, showing no performance gains after that. Lastly, the Largest Empty Circle showed excellent CPU scalability with a consistent decrease in execution times. The performance started slowing down after 12 nodes, which indicated the optimal number of nodes to be used for this problem.

In conclusion, this quarter’s work showed the potential improvements and enhancements that could be made to the current MASS-GIS system. Even though the Euclidean Shortest Path MPI implementation is still incomplete, I am not much behind the project plan’s schedule. I am confident that I can meet the goals and catch up to the project plan by adding more effort and time in the upcoming quarters.

1. **References**

[1] MASS: A parallelizing library for multi-agent spatial simulation. Retrieved July 16, 2023, from <http://depts.washington.edu/dslab/MASS/>

[2] Chuang, T; Fukuda, M. (December 2013). A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems. In Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering. Retrieved July 19, from <https://ieeexplore.ieee.org/abstract/document/6755210>

[3] Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic

foundations and query execution. The VLDB Journal 15, 2, 121–142. Retrieved December 13, from <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/cql.pdf>

[4] Raghavendra, S. P. (2023). Agent-based GIS Queries. University of Washington Bothell. Retrieved October 5, from https://depts.washington.edu/dslab/MASS

[5] United States Department of Transportation, “North American Rail Network Lines,” geodata.bts.gov. The CQL continuous query language: semantic, Retrieved December 13, from

<https://geodata.bts.gov/datasets/d83e85154a304da995837889cc4012e3_0/about>

[6] Kingsford, C. K-d Trees. Carnegie Mellon University. Retrieved December 13, from

<https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf>.c

[7] Alzubaidi, A. M. N. (2021). Minimum Bounding Containers of 2D Convex Polygon. Retrieved December 13, from

<https://www.researchgate.net/publication/354380006_Minimum_Bounding_Containers_of_2D_Convex_Polygon>.

[8] Heineman, G. T., Pollice, G., & Selkow, S. (2016). Algorithms in a Nutshell: A Practical Guide. O’Reilly Media Inc. (Book)

[9] Santoso, H. (2019). Mapping and Grouping of Farm Land with Graham Scan Algorithm on Convex Hull Method. Retrieved December 13, from <https://www.researchgate.net/publication/337508872_Mapping_and_Grouping_of_Farm_Land_with_Graham_Scan_Algorithm_on_Convex_Hull_Method>.

[10] Schuster, M. (2008). The Largest Empty Circle Problem. Retrieved December 13, from <https://www.cs.swarthmore.edu/~adanner/cs97/s08/papers/schuster.pdf>.

[11] Jin, L., Kim, D., Mu, L., Kim, D.-S., & Hu, S.-M. (2006). A Sweepline Algorithm for Euclidean Voronoi Diagram of Circles. Computer-Aided Design, 38(3), 260-272. Retrieved December 13, from <https://cg.cs.tsinghua.edu.cn/papers/cad_2006_voronoi.pdf>.

[12] Foronoi. Retrieved December 13, from <https://voronoi.readthedocs.io/en/latest/>

[13] Coleman, D. (2012). Lee’s O(n2 log n) Visibility Graph Algorithm: Implementation and Analysis. Retrieved December 13, from https://dav.ee/papers/Visibility\_Graph\_Algorithm.pdf.

[14] Martínez, C., & Roura, S. (2001). Optimal Sampling Strategies in Quicksort and Quickselect. SIAM Journal on Computing, 31(3), 683-705. Retrieved December 13, from https://epubs.siam.org/doi/abs/10.1137/S0097539700382108

[15] Preparata, F. P., & Shamos, M. I. (1985). Computational Geometry: An Introduction. Springer Science & Business Media, 2012. Retrieved December 13, from https://www.cs.kent.edu/~dragan/CG/CG-Book.pdf.

[16] Q. Ge, et.al. (2006) “An Improved Algorithm for Finding the Closest Pair of Points,”. Journal of Computer Science and Technology, vol. 21, Issue 1, pp. 27-31. Retrieved December 13, from https://jcst.ict.ac.cn/en/article/pdf/preview/1207.pdf

[17] Kakde, H. M. (2005). Range Searching using Kd Tree. Retrieved December 13, from https://users.cs.utah.edu/~lifeifei/cs6931/kdtree.pdf.

[18] Zhong, C., Malinen, M., Miao, D., Fränti, P. (2014). A fast minimum spanning tree algorithm based on K-means. Information Sciences. Retrieved December 13, from https://iip.tongji.edu.cn/2015INS\_ZCM.pdf

[19] Dijkstra, E. W., Beauguitte, L., Maisonobe, M. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, p. 269271 Version bilingue et commentée. 2021. Retrieved December 13, from <https://dl.acm.org/doi/abs/10.1145/3544585.3544600>

[20] Cui, Y. (2022). Agent-based Graph Applications in MASS Java and Comparison with Spark. University of Washington Bothell. Retrieved December 13, from https://depts.washington.edu/dslab/MASS

**Appendix**

**URL for code**

* https://drive.google.com/drive/folders/115x-qa9ELZEch3DobvhC8Gb7bdNKFxDe?usp=sharing

**How to run the codes**

* Download the Computational Geometry MPI folder from the link.
* Use scp to transfer directory to remote server:
  + scp -r ./Computational Geometry MPI <UW id>@hermes01.uwb.edu:~
* Have java installed.
* If MPI is not already set up, follow instructions from Computational Geometry MPI/range\_search\_mpi/mpi\_setup.txt
* Launch mpd CSSmpdboot -n <number of machines you want to connect to> -v
* To compile the codes:
  + cd <computational geometry problem you want to compile>
  + javac <all the .java files in this folder>
  + e.g. javac KdTree.java Point.java rangeSearch.java
* To run the codes:
  + mpirun -np <number of machines you want to use> java <main class>
  + Range Search: rangeSearch <minX><maxX><minY><maxY>
    - e.g. rangeSearch "23.8647" "49.472737" "-127.663167" "-59.202464"
  + Convex Hull: convexHull
  + Largest Empty Circle: largestCircle
  + Euclidean Shortest Path: euclideanShortest
* Stop all mpds: mpdallexit